闭包（Closures）

1.0 翻译：wh1100717 (https://github.com/wh1100717) 校对：lyuka (https://github.com/lyuka)

2.0 翻译+校对：100mango (https://github.com/100mango)

2.1 翻译：100mango (https://github.com/100mango), magicdict (https://github.com/magicdict) 校对：shanks (http://codebuild.me)

2.2 翻译+校对：SketchK (https://github.com/SketchK) 2016-05-12

本页包含内容：

闭包是自包含的函数代码块，可以在代码中被传递和使用。Swift 中的闭包与 C 和 Objective-C 中的代码块（blocks）以及其他一些编程语言中的匿名函数比较相似。

闭包可以捕获和存储其所在上下文中任意常量和变量的引用。这就是所谓的闭合并包裹着这些常量和变量，俗称闭包。Swift 会为您管理在捕获过程中涉及到的所有内存操作。

> 注意
> 如果您不熟悉捕获（capturing）这个概念也不用担心，您可以在值捕获章节对其进行详细了解。

在函数 (./06_Functions.html)章节中介绍的全局和嵌套函数实际上也是特殊的闭包，闭包采取如下三种形式之一：

- 全局函数是一个有名字但不会捕获任何值的闭包
- 嵌套函数是一个有名字并可以捕获其封闭函数域内值的闭包
- 闭包表达式是一个利用轻量级语法所写的可以捕获其上下文中变量或常量值的匿名闭包

Swift 的闭包表达式拥有简洁的风格，并鼓励在常见场景中进行语法优化，主要优化如下：

- 利用上下文推断参数和返回值类型
- 隐式返回单表达式闭包，即单表达式闭包可以省略 `return` 关键字
- 参数名称缩写
- 尾随（Trailing）闭包语法

闭包表达式（Closure Expressions）

嵌套函数 (./06_Functions.html#nested_function)是一个在较复杂函数中方便进行命名和定义自包含代码模块的方式。当然，有时候撰写小巧的没有完整定义和命名的类函数结构也是很有用处的，尤其是在您处理一些函数并需要将另外一些函数作为该函数的参数时。

闭包表达式是一种利用简洁语法构建内联闭包的方式。闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。下面闭包表达式的例子通过使用几次迭代展示了 `sort(_:)` 方法定义和语法优化的方式。每一次迭代都用更简洁的方式描述了相同的功能。

sort 方法（The Sort Method）

Swift 标准库提供了名为 `sort` 的方法，会根据您提供的用于排序的闭包函数将已知类型数组中的值进行排序。一旦排序完成， `sort(_:)` 方法会返回一个与原数组大小相同,包含同类型元素且元素已正确排序的新数组。原数组不会被 `sort(_:)` 方法修改。

下面的闭包表达式示例使用 `sort(_:)` 方法对一个 `String` 类型的数组进行字母逆序排序.以下是初始数组值：

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

`sort(_:)` 方法接受一个闭包，该闭包函数需要传入与数组元素类型相同的两个值，并返回一个布尔类型值来表明当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 `true` ，反之返回 `false` 。

该例子对一个 `String` 类型的数组进行排序，因此排序闭包函数类型需为 `(String, String) -> Bool` 。

提供排序闭包函数的一种方式是撰写一个符合其类型要求的普通函数，并将其作为 `sort(_:)` 方法的参数传入：

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)
// reversed 为 ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

如果第一个字符串（ `s1` ）大于第二个字符串（ `s2` ）， `backwards(_:_:)` 函数返回 `true` ，表示在新的数组中 `s1` 应该出现在 `s2` 前。对于字符串中的字符来说，“大于”表示“按照字母顺序较晚出现”。这意味着字母 `"B"` 大于字母 `"A"` ，字符串 `"Tom"` 大于字符串 `"Tim"` 。该闭包将进行字母逆序排序， `"Barry"` 将会排在 `"Alex"` 之前。

然而，这是一个相当冗长的方式，本质上只是写了一个单表达式函数（ `a > b` ）。在下面的例子中，利用闭包表达式语法可以更好地构造一个内联排序闭包。

闭包表达式语法（Closure Expression Syntax）

闭包表达式语法有如下一般形式：

```
{ (parameters) -> returnType in
    statements
}
```

闭包表达式语法可以使用常量、变量和 `inout` 类型作为参数，不能提供默认值。也可以在参数列表的最后使用可变参数。元组也可以作为参数和返回值。

下面的例子展示了之前 `backwards(_:_:)` 函数对应的闭包表达式版本的代码：

```
reversed = names.sort({ (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

需要注意的是内联闭包参数和返回值类型声明与 `backwards(_:_:)` 函数类型声明相同。在这两种方式中，都写成了 `(s1: String, s2: String) -> Bool` 。然而在内联闭包表达式中，函数和返回值类型都写在大括号内，而不是大括号外。

闭包的函数体部分由关键字 `in` 引入。该关键字表示闭包的参数和返回值类型定义已经完成，闭包函数体即将开始。

由于这个闭包的函数体部分如此短，以至于可以将其改写成一行代码：

```
reversed = names.sort( { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

该例中 `sort(_:)` 方法的整体调用保持不变，一对圆括号仍然包裹住了方法的整个参数。然而，参数现在变成了内联闭包。

根据上下文推断类型（Inferring Type From Context）

因为排序闭包函数是作为 `sort(_:)` 方法的参数传入的，Swift 可以推断其参数和返回值的类型。 `sort(_:)` 方法被一个字符串数组调用，因此其参数必须是 `(String, String) -> Bool` 类型的函数。这意味着 `(String, String)` 和 `Bool` 类型并不需要作为闭包表达式定义的一部分。因为所有的类型都可以被正确推断，返回箭头（ `->` ）和围绕在参数周围的括号也可以被省略：

```
reversed = names.sort( { s1, s2 in return s1 > s2 } )
```

实际上任何情况下，通过内联闭包表达式构造的闭包作为参数传递给函数或方法时，都可以推断出闭包的参数和返回值类型。 这意味着闭包作为函数或者方法的参数时，您几乎不需要利用完整格式构造内联闭包。

尽管如此，您仍然可以明确写出有着完整格式的闭包。如果完整格式的闭包能够提高代码的可读性，则可以采用完整格式的闭包。而在 `sort(_:)` 方法这个例子里，闭包的目的就是排序。由于这个闭包是为了处理字符串数组的排序，因此读者能够推测出这个闭包是用于字符串处理的。

单表达式闭包隐式返回（Implicit Return From Single-Expression Clossures）

单行表达式闭包可以通过省略 `return` 关键字来隐式返回单行表达式的结果，如上版本的例子可以改写为：

```
reversed = names.sort( { s1, s2 in s1 > s2 } )
```

在这个例子中， `sort(_:)` 方法的参数类型明确了闭包必须返回一个 `Bool` 类型值。因为闭包函数体只包含了一个单一表达式（ `s1 > s2` ），该表达式返回 `Bool` 类型值，因此这里没有歧义， `return` 关键字可以省略。

参数名称缩写（Shorthand Argument Names）

Swift 自动为内联闭包提供了参数名称缩写功能，您可以直接通过 `$0`， `$1`， `$2` 来顺序调用闭包的参数，以此类推。

如果您在闭包表达式中使用参数名称缩写，您可以在闭包参数列表中省略对其的定义，并且对应参数名称缩写的类型会通过函数类型进行推断。 `in` 关键字也同样可以被省略，因为此时闭包表达式完全由闭包函数体构成：

```
reversed = names.sort( { $0 > $1 } )
```

在这个例子中， `$0` 和 `$1` 表示闭包中第一个和第二个 `String` 类型的参数。

运算符函数（Operator Functions）

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。Swift 的 `String` 类型定义了关于大于号（ `>` ）的字符串实现，其作为一个函数接受两个 `String` 类型的参数并返回 `Bool` 类型的值。而这正好与 `sort(_:)` 方法的参数需要的函数类型相符合。因此，您可以简单地传递一个大于号，Swift 可以自动推断出您想使用大于号的字符串函数实现：

```
reversed = names.sort(>)
```

更多关于运算符表达式的内容请查看运算符函数 (./25_Advanced_Operators.html#operator_functions)。

尾随闭包（Trailing Closures）

如果您需要将一个很长的闭包表达式作为最后一个参数传递给函数，可以使用*尾随闭包*来增强函数的可读性。尾随闭包是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用：

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // 函数体部分
}

// 以下是不使用尾随闭包进行函数调用
someFunctionThatTakesAClosure({
    // 闭包主体部分
})

// 以下是使用尾随闭包进行函数调用
someFunctionThatTakesAClosure() {
    // 闭包主体部分
}
```

在闭包表达式语法一节中作为 `sort(_:)` 方法参数的字符串排序闭包可以改写为：

```
reversed = names.sort() { $0 > $1 }
```

如果函数只需要闭包表达式一个参数，当您使用尾随闭包时，您甚至可以把 `()` 省略掉：

```
reversed = names.sort { $0 > $1 }
```

当闭包非常长以至于不能在一行中进行书写时，尾随闭包变得非常有用。举例来说，Swift 的 `Array` 类型有一个 `map(_:)` 方法，其获取一个闭包表达式作为其唯一参数。该闭包函数会为数组中的每一个元素调用一次，并返回该元素所映射的值。具体的映射方式和返回值类型由闭包来指定。

当提供给数组的闭包应用于每个数组元素后， `map(_:)` 方法将返回一个新的数组，数组中包含了与原数组中的元素一一对应的映射后的值。

下例介绍了如何在 `map(_:)` 方法中使用尾随闭包将 `Int` 类型数组 `[16, 58, 510]` 转换为包含对应 `String` 类型的值的数组 `["OneSix", "FiveEight", "FiveOneZero"]`：

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

如上代码创建了一个数字位和它们英文版本名字相映射的字典。同时还定义了一个准备转换为字符串数组的整型数组。

您现在可以通过传递一个尾随闭包给 `numbers` 的 `map(_:)` 方法来创建对应的字符串版本数组：

```
let strings = numbers.map {
    (number) -> String in
    var number = number
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
// strings 常量被推断为字符串类型数组，即 [String]
// 其值为 ["OneSix", "FiveEight", "FiveOneZero"]
```

`map(_:)` 为数组中每一个元素调用了闭包表达式。您不需要指定闭包的输入参数 `number` 的类型，因为可以通过要映射的数组类型进行推断。

在该例中，局部变量 `number` 的值由闭包中的 `number` 参数获得,因此可以在闭包函数体内对其进行修改，(闭包或者函数的参数总是固定的),闭包表达式指定了返回类型为 `String` ，以表明存储映射值的新数组类型为 `String` 。

闭包表达式在每次被调用的时候创建了一个叫做 `output` 的字符串并返回。其使用求余运算符（ `number % 10` ）计算最后一位数字并利用 `digitNames` 字典获取所映射的字符串。

> 注意
> 字典 `digitNames` 下标后跟着一个叹号（ `!` ），因为字典下标返回一个可选值（optional value），表明该键不存在时会查找失败。在上例中，由于可以确定 `number % 10` 总是 `digitNames` 字典的有效下标，因此叹号可以用于强制解包 (force-unwrap) 存储在下标的可选类型的返回值中的 `String` 类型的值。

从 `digitNames` 字典中获取的字符串被添加到 `output` 的前部，逆序建立了一个字符串版本的数字。（在表达式 `number % 10` 中，如果 `number` 为 `16` ，则返回 `6` ， `58` 返回 `8` ， `510` 返回 `0` 。）

`number` 变量之后除以 `10` 。因为其是整数，在计算过程中未除尽部分被忽略。因此 `16` 变成了 `1` ， `58` 变成了 `5` ， `510` 变成了 `51` 。

整个过程重复进行，直到 `number /= 10` 为 `0` ，这时闭包会将字符串 `output` 返回，而 `map(_:)` 方法则会将字符串添加到所映射的数组中。

在上面的例子中，通过尾随闭包语法，优雅地在函数后封装了闭包的具体功能，而不再需要将整个闭包包裹在 `map(_:)` 方法的括号内。

捕获值（Capturing Values）

闭包可以在其被定义的上下文中*捕获*常量或变量。即使定义这些常量和变量的原作用域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift 中，可以捕获值的闭包的最简单形式是嵌套函数，也就是定义在其他函数的函数体内的函数。嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

举个例子，这有一个叫做 `makeIncrementor` 的函数，其包含了一个叫做 `incrementor` 的嵌套函数。嵌套函数 `incrementor()` 从上下文中捕获了两个值， `runningTotal` 和 `amount` 。捕获这些值之后， `makeIncrementor` 将 `incrementor` 作为闭包返回。每次调用 `incrementor` 时，其会以 `amount` 作为增量增加 `runningTotal` 的值。

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

`makeIncrementor` 返回类型为 `() -> Int` 。这意味着其返回的是一个函数，而不是一个简单类型的值。该函数在每次调用时不接受参数，只返回一个 `Int` 类型的值。关于函数返回其他函数的内容，请查看函数类型作为返回类型 (./06_Functions.html#function_types_as_return_types)。

`makeIncrementer(forIncrement:)` 函数定义了一个初始值为 `0` 的整型变量 `runningTotal` ，用来存储当前跑步总数。该值通过 `incrementor` 返回。

`makeIncrementer(forIncrement:)` 有一个 `Int` 类型的参数，其外部参数名为 `forIncrement` ，内部参数名为 `amount` ，该参数表示每次 `incrementor` 被调用时 `runningTotal` 将要增加的量。

嵌套函数 `incrementor` 用来执行实际的增加操作。该函数简单地使 `runningTotal` 增加 `amount` ，并将其返回。

如果我们单独看这个函数，会发现看上去不同寻常：

```
func incrementor() -> Int {
    runningTotal += amount
    return runningTotal
}
```

`incrementer()` 函数并没有任何参数，但是在函数体内访问了 `runningTotal` 和 `amount` 变量。这是因为它从外围函数捕获了 `runningTotal` 和 `amount` 变量的引用。捕获引用保证了 `runningTotal` 和 `amount` 变量在调用完 `makeIncrementer` 后不会消失，并且保证了在下一次执行 `incrementer` 函数时， `runningTotal` 依旧存在。

> 注意
> 为了优化，如果一个值是不可变的，Swift 可能会改为捕获并保存一份对值的拷贝。
> Swift 也会负责被捕获变量的所有内存管理工作，包括释放不再需要的变量。

下面是一个使用 `makeIncrementor` 的例子：

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

该例子定义了一个叫做 `incrementByTen` 的常量，该常量指向一个每次调用会将 `runningTotal` 变量增加 `10` 的 `incrementor` 函数。调用这个函数多次可以得到以下结果：

```
incrementByTen()
// 返回的值为10
incrementByTen()
// 返回的值为20
incrementByTen()
// 返回的值为30
```

如果您创建了另一个 `incrementor` ，它会有属于它自己的一个全新、独立的 `runningTotal` 变量的引用：

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// 返回的值为7
```

再次调用原来的 `incrementByTen` 会在原来的变量 `runningTotal` 上继续增加值，该变量和 `incrementBySeven` 中捕获的变量没有任何联系：

```
incrementByTen()
// 返回的值为40
```

> 注意
> 如果您将闭包赋值给一个类实例的属性，并且该闭包通过访问该实例或其成员而捕获了该实例，您将创建一个在闭包和该实例间的循环强引用。Swift 使用捕获列表来打破这种循环强引用。更多信息，请参考闭包引起的循环强引用 (./16_Automatic_Reference_Counting.html#strong_reference_cycles_for_closures)。

闭包是引用类型（Closures Are Reference Types）

上面的例子中， `incrementBySeven` 和 `incrementByTen` 是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量的值。这是因为函数和闭包都是*引用类型*。

无论您将函数或闭包赋值给一个常量还是变量，您实际上都是将常量或变量的值设置为对应函数或闭包的引用。上面的例子中，指向闭包的引用 `incrementByTen` 是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量或变量，两个值都会指向同一个闭包：

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// 返回的值为50
```

非逃逸闭包(Nonescaping Closures)

当一个闭包作为参数传到一个函数中，但是这个闭包在函数返回之后才被执行，我们称该闭包从函数中*逃逸*。当你定义接受闭包作为参数的函数时，你可以在参数名之前标注 `@noescape` ，用来指明这个闭包是不允许"逃逸"出这个函数的。将闭包标注 `@noescape` 能使编译器知道这个闭包的生命周期（译者注：闭包只能在函数体中被执行，不能脱离函数体执行，所以编译器明确知道运行时的上下文），从而可以进行一些比较激进的优化。

```
func someFunctionWithNoescapeClosure(@noescape closure: () -> Void) {
    closure()
}
```

举个例子， `sort(_:)` 方法接受一个用来进行元素比较的闭包作为参数。这个参数被标注了 `@noescape` ，因为它确保自己在排序结束之后就没用了。

一种能使闭包"逃逸"出函数的方法是，将这个闭包保存在一个函数外部定义的变量中。举个例子，很多启动异步操作的函数接受一个闭包参数作为 completion handler。这类函数会在异步操作开始之后立刻返回，但是闭包直到异步操作结束后才会被调用。在这种情况下，闭包需要"逃逸"出函数，因为闭包需要在函数返回之后被调用。例如：

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: () -> Void) {
    completionHandlers.append(completionHandler)
}
```

`someFunctionWithEscapingClosure(_:)` 函数接受一个闭包作为参数，该闭包被添加到一个函数外定义的数组中。如果你试图将这个参数标注为 `@noescape` ，你将会获得一个编译错误。

将闭包标注为 `@noescape` 使你能在闭包中隐式地引用 `self` 。

```
class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNoescapeClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// prints "200"

completionHandlers.first?()
print(instance.x)
// prints "100"
```

自动闭包（Autoclosures）

*自动闭包*是一种自动创建的闭包，用于包装传递给函数作为参数的表达式。这种闭包不接受任何参数，当它被调用的时候，会返回被包装在其中的表达式的值。这种便利语法让你能够用一个普通的表达式来代替显式的闭包，从而省略闭包的花括号。

我们经常会调用一个接受闭包作为参数的函数，但是很少实现那样的函数。举个例子来说， `assert(condition:message:file:line:)` 函数接受闭包作为它的 `condition` 参数和 `message` 参数；它的 `condition` 参数仅会在 debug 模式下被求值，它的 `message` 参数仅当 `condition` 参数为 `false` 时被计算求值。

自动闭包让你能够延迟求值，因为代码段不会被执行直到你调用这个闭包。延迟求值对于那些有副作用（Side Effect）和代价昂贵的代码来说是很有益处的，因为你能控制代码什么时候执行。下面的代码展示了闭包如何延时求值。

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// prints "5"

let customerProvider = { customersInLine.removeAtIndex(0) }
print(customersInLine.count)
// prints "5"

print("Now serving \(customerProvider())!")
// prints "Now serving Chris!"
print(customersInLine.count)
// prints "4"
```

尽管在闭包的代码中， `customersInLine` 的第一个元素被移除了，不过在闭包被调用之前，这个元素是不会被移除的。如果这个闭包永远不被调用，那么在闭包里面的表达式将永远不会执行，那意味着列表中的元素永远不会被移除。请注意， `customerProvider` 的类型不是 `String` ，而是 `() -> String` ，一个没有参数且返回值为 `String` 的函数。

将闭包作为参数传递给函数时，你能获得同样的延时求值行为。

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serveCustomer(customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serveCustomer( { customersInLine.removeAtIndex(0) } )
// prints "Now serving Alex!"
```

`serveCustomer(_:)` 接受一个返回顾客名字的显式的闭包。下面这个版本的 `serveCustomer(_:)` 完成了相同的操作，不过它并没有接受一个显式的闭包，而是通过将参数标记为 `@autoclosure` 来接收一个自动闭包。现在你可以将该函数当做接受 `String` 类型参数的函数来调用。 `customerProvider` 参数将自动转化为一个闭包，因为该参数被标记了 `@autoclosure` 特性。

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serveCustomer(@autoclosure customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serveCustomer(customersInLine.removeAtIndex(0))
// prints "Now serving Ewa!"
```

> 注意
>
> 过度使用 `autoclosures` 会让你的代码变得难以理解。上下文和函数名应该能够清晰地表明求值是被延迟执行的。

`@autoclosure` 特性暗含了 `@noescape` 特性，这个特性在非逃逸闭包一节中有描述。如果你想让这个闭包可以“逃逸”，则应该使用 `@autoclosure(escaping)` 特性.

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(@autoclosure(escaping) customerProvider: () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.removeAtIndex(0))
collectCustomerProviders(customersInLine.removeAtIndex(0))

print("Collected \(customerProviders.count) closures.")
// prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// prints "Now serving Barry!"
// prints "Now serving Daniella!"
```

在上面的代码中， `collectCustomerProviders(_:)` 函数并没有调用传入的 `customerProvider` 闭包，而是将闭包追加到了 `customerProviders` 数组中。这个数组定义在函数作用域范围外，这意味着数组内的闭包将会在函数返回之后被调用。因此， `customerProvider` 参数必须允许“逃逸”出函数作用域。

上一篇: 函数 (/project/swift/chapter2/06_Functions.html)

下一篇: 枚举 (/project/swift/chapter2/08_Enumerations.html)

被顶起来的评论

NewProgrammer (http://www.baidu.com/p/NewProgrammer)

非逃逸闭包那部分请教各位是什么意思，原话“当一个闭包作为参数传到一个函数中，但是这个闭包在函数返回之后才被执行，我们称该闭包从函数中逃逸。”是指默认写的闭包是逃逸闭包吗？就是说如果不添加“@noescape”的话，闭包的内容会在函数结束后调用?
我照案例做了小段代码试验了下，发现，添加“@noescape”和没有添加没有区别，都是先执行的闭包而后执行函数体。代码如下:

<code>
func doSometing(@noescape some:() -> Void){
some()
print("函数体")
}
doSometing({
print("逃逸闭包")
})
执行结果如下:

逃逸闭包
函数体

<code/>

2015年11月23日 回复 顶(2) 转发

Ken

回复 NewProgrammer: 同步操作的话，是没有体现出noescape的特别，主要是在异步的时候起作用，譬如，你在doSomething里面执行异步的网络请求，你需要用some来执行一些请求后的操作的话，如果用noescape的话，会变异出错，请求的返回处理some函数是在doSomething这个函数结束后才会执行，根据noescape的定义，doSomething结束后就不能执行some了，你可以在xcode7.2上面测试下面的代码，估计会在some()上面会报错

```
func doSometing(@noescape some:() -> Void){

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (Int64)(2 * NSEC_PER_SEC)),
dispatch_get_main_queue()) { () -> Void in
some()
}
print("函数体")
}
doSometing({
print("逃逸闭包")
})
```

2月1日 回复 顶(1) 转发

G梅果果 (http://weibo.com/2063322140)

回复 NewProgrammer: 非逃逸闭包和逃逸闭包讲的不是执行先后顺序吧,非逃逸是指你那个 some 闭包不能在函数外单独调用,只能在函数内部调用,函数调用完成后,那个闭包也就结束了,而逃逸闭包,你看官方给的例子,将闭包加入了数组,通过在外部访问数组元素,从而达到了使用那个闭包的效果

"someFunctionWithEscapingClosure(_:)函数接受一个闭包作为参数，该闭包被添加到一个函数外定义的数组中。如果你试图将这个参数标注为@noescape，你将会获得一个编译错误。"

2015年12月18日 回复 顶(1) 转发

38 条评论 最新 最早 最热

Don

闭包的参数名和局部变量名可以重名?

7月9日 回复 顶 转发

Ace (http://t.qq.com/flash_boy)

闭包这一章对于初学者来说，真是很难懂

6月14日 回复 顶 转发

路熏 (http://t.qq.com/l592816909)

看到这里有些吃力了

5月5日 回复 顶 转发

周小明

单表达式闭包隐式返回（Implicit Return From Single-Expression Clossures）Clossures 错了

3月4日 回复 顶 转发

Ken

回复 NewProgrammer: 同步操作的话，是没有体现出noescape的特别，主要是在异步的时候起作用，譬如，你在doSomething里面执行异步的网络请求，你需要用some来执行一些请求后的操作的话，如果用noescape的话，会变异出错，请求的返回处理some函数是在doSomething这个函数结束后才会执行，根据noescape的定义，doSomething结束后就不能执行some了，你可以在xcode7.2上面测试下面的代码，估计会在some()上面会报错

```
func doSometing(@noescape some:() -> Void){

dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (Int64)(2 * NSEC_PER_SEC)),
dispatch_get_main_queue()) { () -> Void in
some()
}
print("函数体")
}
doSometing({
print("逃逸闭包")
})
```

2月1日 回复 顶(1) 转发

aMong

回复 NewProgrammer: 逃逸闭包是外部是可以访问的 非逃逸闭包只能在函数体内部访问 也就是函数体执行结束之后非逃逸闭包也就失效了

1月27日 回复 顶 转发

贾陆华 (http://t.qq.com/microbyte)

回复 徐健哲: 闭包逃逸和非逃逸看起来就是闭包生命周期的范围问题；尾随闭包和自动闭包这些都是语法糖了，让代码更具可读性

1月15日 回复 顶 转发

(http://www.h2byte.com/) 特立独行猪 (http://www.h2byte.com/)
回复 Robert Li: z既然是全局变量，那么肯定是可以被任何函数访问的。“可以访问”和“捕获”两者之间还是有区别的。文中runningTotal是incrementBySeven()和incrementByTen()捕获到的变量，两个runningTotal互不影响

1月11日 回复 顶 转发

芒果

回复 NewProgrammer: 你的例子没看出逃逸闭包的用法,逃逸闭包是指在函数体执行之后,才在另外一个时间执行的闭包,在你的doSometing里面,已经明显的调用了some()这个闭包了,哪里还是逃逸呢?

1月8日 回复 顶 转发


(http://weibo.com/wenbokenet) dong (http://weibo.com/wenbokenet)
将闭包标注为@noescape使你能在闭包中隐式地引用self。想表达什么意思呢，反正都要调self，只不过可以省略，省略的话，就是这里x的调用是唯一的，没有二义性，逃逸出去的闭包，必须显式调用self，也就是说，可能出现同名的标识符，这里说的有点含糊啊，同学们给讨论讨论呗

1月3日 回复 顶 转发


(http://weibo.com/1725480263) kakapo泓 (http://weibo.com/1725480263)
回复 dddd: 你的意思是说 用汇编的是最高级的？ 逗比。这思想不错

1月3日 回复 顶 转发


(http://t.qq.com/xclidongbo) 李东波 (http://t.qq.com/xclidongbo)
回复 G梅果果: 这个说得很好

2015年12月29日 回复 顶 转发

Hugo

回复 NewProgrammer: 如果闭包需要赋值给函数外部变量或常量的话，就不能使用非逃逸闭包

2015年12月21日 回复 顶 转发

Lay

回复 NewProgrammer: 你在 doSomething中执行闭包 意味着将它当做非逃逸闭包处理了 所以你这样是没有区别的 逃逸闭包不代表不能在函数体中执行。逃逸闭包与非逃逸闭包的区别在于 逃逸闭包 可以在函数体之外执行，比如例子中的 在函数体中将闭包放入外部的数组中 在外面调用 array[index]() 去执行它。 而标记为非逃逸闭包之后 "你只能在接受闭包的函数体中执行这个闭包" 而不能向逃逸闭包那样 先在接收函数中放入数组 在函数外去执行它

2015年12月21日 回复 顶 转发


(http://weibo.com/2063322140) G梅果果 (http://weibo.com/2063322140)
回复 NewProgrammer: 非逃逸闭包和逃逸闭包讲的不是执行先后顺序吧,非逃逸是指你那个 some 闭包不能在函数外单独调用,只能在函数内部调用,函数调用完成后,那个闭包也就结束了,而逃逸闭包,你看官方给的例子,将闭包加入了数组,通过在外部访问数组元素,从而达到了使用那个闭包的效果
"someFunctionWithEscapingClosure(_:)函数接受一个闭包作为参数，该闭包被添加到一个函数外定义的数组中。如果你试图将这个参数标注为@noescape，你将会获得一个编译错误。"

2015年12月18日 回复 顶(1) 转发


(http://weibo.com/ndoc) 黄帝的新医 (http://weibo.com/ndoc)
回复 xiao-律-fighting: 难道不应该是 ["Ewa", "Daniella", "Chris", "Barry", "Alex"] 吗？ 没有错啊

2015年12月13日 回复 顶 转发

dddd

回复 沉沦2013: 额,这有什么,用的语言越高级,程序员越低级

2015年12月9日 回复 顶 转发


(http://weibo.com/caoping) caoping (http://weibo.com/caoping)
回复 NewProgrammer: 我理解的是，@noescape这个标记除了隐式self之外不会有其他作用，它其实是为了告诉函数调用者，表示这个闭包参数会在什么时候执行，也就是同步还是异步执行。同步就是在函数返回前就调用闭包，异步就是我先hold住这个闭包，然后函数立刻返回，等需要的时候再调用这个闭包。类似oc中的block参数，这个种逃逸闭包在网络请求中最常用了

2015年12月8日 回复 顶 转发

Robert Li

```
//关于上文中指出：全局函数是一个有名字但不会捕获任何值的闭包
//我个人的理解是：全局函数是一个有名字并可以捕获到全局作用域内值的闭包

var z=0;//全局作用域内的值
func testFun(){//全局函数
z++
}

func funMaker() -> ()->(Void){
return testFun
}
```

```
var fun1 = funMaker()
//一下用于执行输出结果
fun1()
print(z);//输出1
fun1()
print(z);//输出2
```

2015年11月29日 回复 顶 转发

李东波 (http://t.qq.com/xclidongbo)
还是看不明白,你说的逃逸函数是啥意思.
(http://t.qq.com/xclidongbo)2015年11月28日 回复 顶 转发